Week 8 : Numpy

pending tasks	
type	

Numpy

Numpy is used to work (storing and processing) with high dimensional numerical arrays efficiently. Numpy stores homogenous data types.

Lists can store and process collection of high dimensional numbers as arrays through iteration but this is very inefficient.

- Lists are designed to store heterogeneous data, thus, has an overhead of type checking.
- Lists do not exploit low level hardware mechanisms like DRAM burst access and vectorisation.

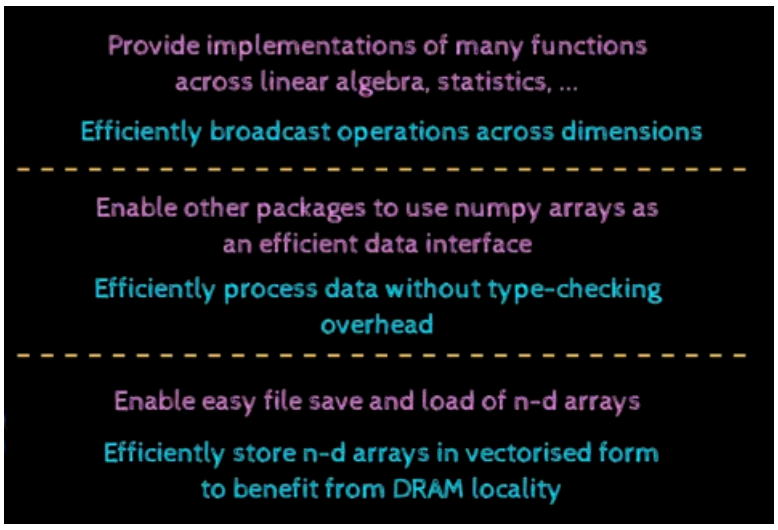


Fig.8.1 Hardware and OS-level, programming language-level, user perspective

Comparing performance of Numpy with lists

```
N = 100000000
# using iteration
%%time
list_ = list(range(N))
for i in range(N):
    list_[i] = list_[i] * list_[i]
>>> CPU times: user 18.3 s, sys: 1.52 s, total: 19.8 s Wall time: 19.8 s
# using list comprehension
list_ = list(range(N))
list_ = [item * item for item in list_]
>>> CPU times: user 9.14 s, sys: 3.74 s, total: 12.9 s Wall time: 12.9 s
# using map() and lambda function
list_ = list(range(N))
list_ = map(lambda x: x * x, list_)
>>> CPU times: user 2.38 s, sys: 1.43 s, total: 3.81 s Wall time: 3.82 s
```

There are always ways to optimise the code, it is best to use an in-built function rather than defining the same. The example below shows addition of elements of a list.

```
# using iteration
list_ = list(range(N))
list_sum = 0
for item in list_:
    list_sum += item
>>> CPU times: user 12.2 s, sys: 1.36 s, total: 13.6 s Wall time: 13.6 s
# using inbuilt sum() function
list_ = list(range(N))
list_sum = sum(list_)
>>> CPU times: user 2.71 s, sys: 1.34 s, total: 4.05 s Wall time: 4.04 s
```

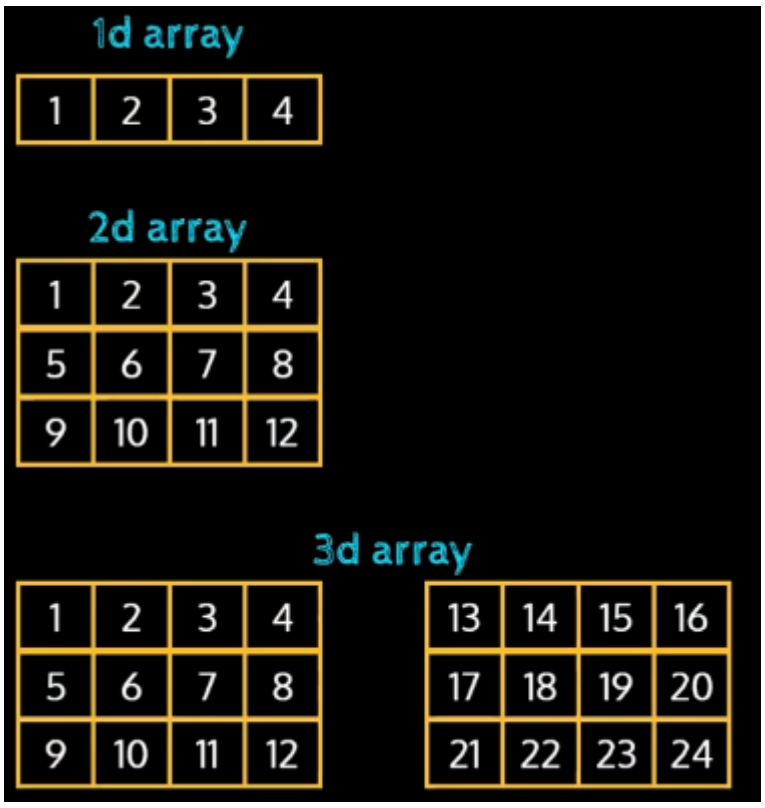
Using numpy :

```
# creating a numpy array and squaring it
arr = np.arange(N)
```

```
arr = arr * arr
>>> CPU times: user 349 ms, sys: 5.05 ms, total: 354 ms Wall time: 365 ms
# sum of n numbers using numpy
arr = np.arange(N)
arr_sum = np.sum(arr)
>>> CPU times: user 235 ms, sys: 16 ms, total: 251 ms Wall time: 252 ms
```

Numpy works significantly faster because of the way it is stored, iterated over. Numpy is implemented in C.

High dimensional array & creating numpy array



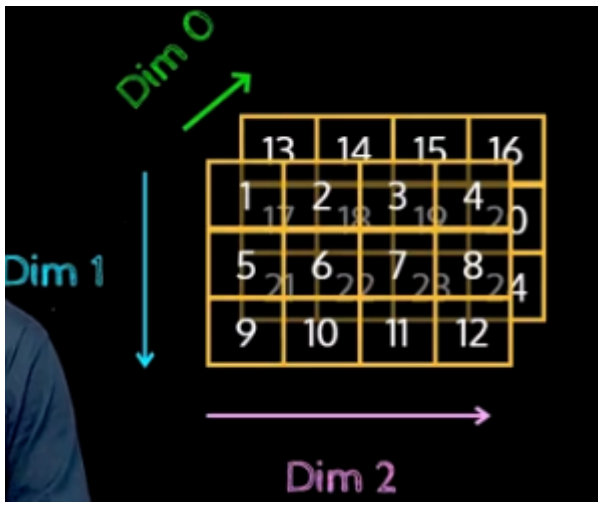


Fig.8.2 understanding high dimensional arrays using array A.

- The array dimensions are indexed backwards (with respect to the order they were added).e.g. The shape of the array A in the fig.8.2 is 2×3×4 (two planes that were added last - three rows in each plan - four columns in each row which were added first).
- The numpy arrays can be sliced .e.g all the elements of plan 0 of array A can be selected using A[0,:,:].
- The slices can be referred using partial indices like in fig.8.3. The output of indexing is also a numpy array.

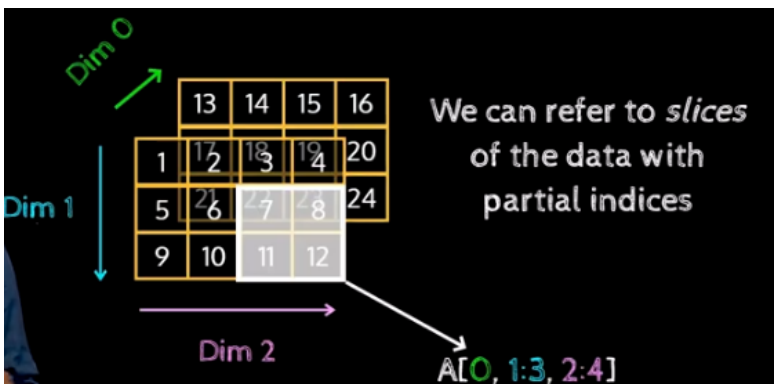


Fig.8.3 referring to slices of array A using partial indices.

Creating numpy arrays:

```
# using arange
arr = np.arange(5) # forms an array with numbers 0-4

# using a list to create np array
arr = np.array([0.0, 2, 4, 6, 8])

#creating 2D array
arr2d = np.array([[1, 2, 3],
                  [4, 5, 6]])

#creating 3D array
arr3d = np.array([[[1, 2, 3],
                   [4, 5, 6]],
                  [[7, 8, 9],
                   [10, 11, 12]]])

#creating array with constant value
```

```
1729* np.ones((3, 4))
>>> array([[1729., 1729., 1729., 1729.],
           [1729., 1729., 1729., 1729.],
           [1729., 1729., 1729., 1729.]])

# creating arrays with all zeros
np.zeros((3, 4))
>>> array([[0., 0., 0., 0.],
           [0., 0., 0., 0.],
           [0., 0., 0., 0.]])

# creating arrays using random values
np.random.randn(2,3) # mean 0 and std 1 - normal distribution
>>> array([[ 1.42970912, -0.07279769,  1.3693901 ],
          [-1.51340464,  0.72139968,  0.70462965]])

# rand - uniformly sampled numbers between 0,1
np.random.rand(2, 3)
>>> array([[0.17890542, 0.71095944, 0.77045343],
          [0.71310117, 0.1987117 , 0.3749793 ]])

# randint - start num, end num, shape of integer array
np.random.randint(0, 100, (2, 3))
>>>array([[18, 25, 78],
          [97, 86,  6]])

# using arange(lower_no,high_no(not included) ,step_size)
np.arange(7, 71, 7)
>>> array([ 7, 14, 21, 28, 35, 42, 49, 56, 63, 70])

# linear space
np.linspace(7, 70, 10): the high_no is included
>>> array([ 7., 14., 21., 28., 35., 42., 49., 56., 63., 70.])
```

- Numpy arrays can be created for boolean and strings too not just numeric types.
- String arrays with numbers can be converted into numeric type by specifying the dtype.

```
str_arr = np.array(['1.4', '2.1', '1.1'])
arr = np.array(str_arr, dtype='float')
```

- Checking attributes of array

```
# datatype
arr.dtype
>>> dtype('float64')
# dimensions
arr.ndim
>>> 1
#shape - shape of each dimension as a tuple
arr.shape
>>> (5,)
#size - total number of elements in the array
arr.size
>>> 5
# itemsize - number of bytes required to store each element
arr.itemsize
>>> 8
```

Indexing

- Indexing is used to access the elements of the arrays.
- Single elements of the array can be accessed using their index in the array. And slices of numbers can be accessed using colon (:) .

```
arr3d = np.array([[[1, 2, 3],
                   [4, 5, 6]],
                  [[7, 8, 9],
                   [10, 11, 12]]])
# accessing single elements
arr3d[1, 0, 2]
>>> 9
# accessing slices
arr3d[1, :, :]
>>> array([[ 7,  8,  9],
           [10, 11, 12]])
```

- Slicing returns a numpy array(shallow copy) which can be further sliced.
- Fancy indexing can be done when the position of the element is not known. This involves writing conditional statements that a given element must satisfy.

```
#fancy indexing
arr3d % 2 == 0
>>> array([[[False,  True, False],
            [True, False,  True]],
           [[False,  True, False],
            [ True, False,  True]]])
arr3d[(arr3d % 2 == 1) & (arr3d > 3)]
>>> array([ 5,  7,  9, 11])
```

- A list containing indices of elements to be accessed can be used.

```
# indexing using lists of indices
my_indices = [1, 3, 4]
arr[my_indices]
>>> array([8, 1, 8])

# creating deep copy of a numpy array
arr_slice = np.copy(arr3d[:, :, 0:2])
```

Numpy operations

- The operations can be done considering the numpy objects as vectors. Point wise add,sub, mul and div can be done this way.
- Log, exponentiation, trigonometric functions, square root etc. can also be done.

```
# example operations on numpy arrays
arr1 = np.random.rand(3, 4)
print(arr1)
>>> array([[0.22962077, 0.51507481, 0.73337689, 0.0732048 ],
           [0.84785699, 0.1344861 , 0.41097148, 0.79916757],
           [0.29367862, 0.72556205, 0.33804899, 0.70370708]])
arr2 = np.random.rand(3, 4)
arr1 + arr2 # + op can be replaced by -,*,/
>>> array([[1.21771204, 0.60189773, 1.63035461, 0.58971738],
           [1.57854769, 0.40674081, 0.97613881, 1.64804876],
           [0.32302657, 1.37565014, 0.72627243, 1.45083187]])
np.exp(arr1) # sin,cos,sqrt functions can be used this way
>>> array([[1.2581228 , 1.67376371, 2.08209977, 1.07595087],
           [2.33463834, 1.14394876, 1.50828234, 2.22368909],
           [1.34135275, 2.06589191, 1.4022092 , 2.0212317 ]])
np.log(np.exp(arr1))
>>> array([[0.22962077, 0.51507481, 0.73337689, 0.0732048 ],
           [0.84785699, 0.1344861 , 0.41097148, 0.79916757],
           [0.29367862, 0.72556205, 0.33804899, 0.70370708]])
```

- Division by zero does not throw an error it returns the keyword inf.

```
arr1 = np.zeros((3, 5))
arr_inv = 1/arr1
np.isinf(arr_inv[0,0])
>>> True
```

Question:

Exercise on finding the number of points outside n-dimensional space.

- Use np.random.rand() to create a matrix of (npoints*ndim). This creates a random uniformly distributed sample npoints of ndim between (0,1)
- Calculate the distance from origin for each point. Using iteration takes longer time as it cannot be vectorised. Thus, use numpy methods.

```
sq_points = points * points
dfo = np.sqrt(np.sum(sq_points, axis = 1))
outside_point = np.sum(dfo>1)
```

- Check the fraction of points with distance >1
- Single line of code is faster as no memory allocation task is explicitly carried out.
- Higher the dimension, more would be the number of points that lie away from the origin.

Broadcasting

- Lower dimensions are broadcasted along the higher dimensions.
- The values are broadcasted (copied) along a dimension, if the corresponding dimension is empty or 1. The dimensions are compared from left to right.
- Broadcasting works when the corresponding dimension values exactly map, are missing or 1.

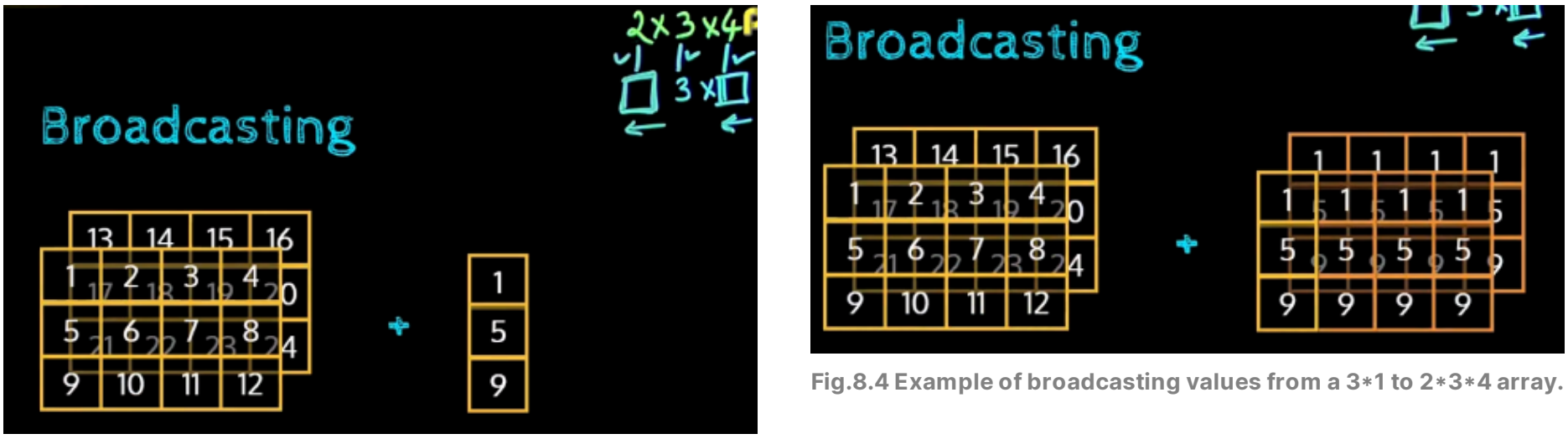


Fig.8.4 Example of broadcasting values from a 3*1 to 2*3*4 array.

- arr.reshape((dimension,)) can be used to modify the shape of the array (row major order by default).

```
arr1 = np.arange(6)
arr1 = arr1.reshape((3, 2))
arr2 = np.arange(6).reshape((3, 2))
arr1 + arr2[0].reshape((1, 2)) # (3, 2) + (1, 2)
>>>  array([[0, 2],
            [2, 4],
            [4, 6]])
arr1 = np.arange(4)
arr2 = np.arange(5)
arr1.T + arr2 # same as arr1.reshape(4,1)
>>> array([[0, 1, 2, 3, 4],
       [1, 2, 3, 4, 5],
       [2, 3, 4, 5, 6],
       [3, 4, 5, 6, 7]])
```

File handling

- Numpy requires homogenous data types, thus row names and column names if present have to be slipped while reading a file as np array.

```
planets_small = np.loadtxt("planets_small.txt",
                           skiprows = 1,
                           usecols = (1, 2, 3, 4, 5, 6, 7, 8, 9))
```

- use np.genfromtxt() to parse the unknown and blank values into numpy compatible types. These values are filled using **nan** values.

```
planets = np.genfromtxt("planets.txt",
                        skip_header=1,
```

```
                        usecols = [1, 2, 3, 4, 5, 6, 7, 8, 9])
np.isnan(planets) # returns true for values which are nan
```

- nan can be encoded using a number that does not occur in the data as a flag.

```
planets_new = np.nan_to_num(planets, nan=-1)
```

- The numpy arrays can be saved as txt files or numpy files (binary). The file size of numpy files is smaller than txt files. In the example below, the txt is of 4.5 kB whereas the numpy file is of 1.6kB.

```
np.savetxt('planets_new.txt', planets_new,
           delimiter=',')
np.save("planets_new", planets_new)
```

- Multiple arrays can be saved into a numpy file and read back as individual arrays from the file.

```
arr1 = np.random.rand(1000, 10)
arr2 = np.random.rand(2000, 5)
arr3 = np.random.rand(20, 10000)

np.savez("many_arrs", arr1, arr2, arr3)
arrs = np.load('many_arrs.npz')
arrs.files
>>> ['arr_0', 'arr_1', 'arr_2']
arrs['arr_0'].shape
>>> (1000, 10)
```

- Numpy supports storing data in a compressed data. This is useful while storing sparse data matrix. In the example below, the size of uncompressed file is 760 mB while the compressed file is 760 kB.

```
arr1 = np.zeros((10000, 10000))
np.savez("zeros", arr1)
np.savez_compressed("zeros_compressed", arr1)
```

Stats with numpy

def z-score - (t = 7:30)

- Common stats functions amin(),amax(),mean(),median(),std(),var() can be calculated. The syntax is **np.stat_fn(np_array).**
- It is better to use lesser calls to numpy from the user end. This would improve the performance. Given below is the example to calculate the IQR.

```
%%time
iqr = np.percentile(arr, 75) - np.percentile(arr, 25)
>>> CPU times: user 280 ms, sys: 36 ms, total: 316 ms Wall time: 319 ms
quartiles = np.percentile(arr, [25, 75])
iqr = quartiles[1] - quartiles[0]
>>> CPU times: user 210 ms, sys: 0 ns, total: 210 ms Wall time: 210 ms
```

- z-score tells how far away a given point is from the mean, given in terms of standard deviation. Negative number implies that the point lies in the LHS and a positive number indicates that it lies in the RHS of the mean.

```
(arr - np.mean(arr))/np.std(arr)
```

- Histograms can be derived from data in the form of two arrays. One is the bins and the other is the number of members in each bin. If bins is not defined, then it is implicitly calculated for the given array.

```
np.histogram(arr, bins=[0, 0.25, 0.5, 0.75, 1])
>>> (array([2497818, 2498813, 2500099, 2503270]),
     array([0.  , 0.25, 0.5 , 0.75, 1.  ]))
```

- Mapping of a point to a bin can be found using the digitize function. In the bins, the left point is included and the right point is excluded. This can be changed by setting right=True in digitize function.

```
arr1 = np.random.randint(0, 10, (10))
arr1
>>>array([2, 8, 8, 2, 6, 0, 8, 2, 7, 9])
bins = [0, 6, 10]
np.digitize(arr1, bins)
>>> array([1, 2, 2, 1, 2, 1, 2, 1, 2, 2])
```

- Different arrays of numpy can be concatenated along a particular axis using functions like vstack and hstack,column_stack. These are implementations of the np.concatenate function.

```
arr1 = np.random.randint(50, 80, 100) # weight
arr2 = np.random.randint(150, 185, 100) # heights
arr3 = np.random.randint(17, 22, 100) # age
arr2D = np.vstack((arr1, arr2, arr3))
arr2D.shape
>>> (3, 100)
```

- The stat functions can be calculated along a given axis.

```
np.amin(arr2d, axis=1)
>>> array([ 50, 150,  17])
```

Rules of statistics

- Mean subtracted array has zero mean.
- Computing the mean from a smaller number of values can be biased.

```
means = np.cumsum(arr)/np.arange(1,1001)
means[0:15]
>>> array([0.80334668, 0.47415025, 0.50657624, 0.57224576, 0.6124674 ,
           0.51811314, 0.5048491 , 0.51491554, 0.49235724, 0.54249526,
           0.52085287, 0.48670218, 0.48395448, 0.45357307, 0.48506273])
```

- Mean is sensitive to outliers whereas the median is more robust. Values can be added to a numpy array using np.append() method.

```
arr = np.append(arr, [1000, 2000])
```

- Scaling arrays results in scaling and shifting the mean and the median by the same amount. Standard deviation is scaled by ‘a’, variance by a2, there will be no shift.

```
print(np.mean(2.5 * arr + 0.65), 2.5 * np.mean(arr) + 0.65)
>>> 1.7457826649755248 1.7457826649755246

print(np.median(2.5 * arr + 0.65), 2.5 * np.median(arr) + 0.65)
>>> 1.6425977090487114 1.6425977090487116

print(np.std(2.5 * arr + 0.65), 2.5 * np.std(arr))
>>> 0.6453129604733241 0.6453129604733241

print(np.var(2.5 * arr + 0.65), 2.5*2.5*np.var(arr))
>>> 0.4164288169548459 0.4164288169548459

print(np.mean(0.21*arr1 - 0.75 * arr2),0.21*np.mean(arr1)-0.75*np.mean(arr2))
>>> -0.29677089842238513 -0.29677089842238513
```

Summary

- Numpy is a python library implemented in C, efficient in manipulation of high dimensional arrays.
- The dimensions of a numpy array are numbered in the reverse order of addition.

- Native numpy operations are faster than using loops.
- Broadcasting can be done when the corresponding dimensions of the two numpy arrays match, either is 1 or 0 when taken from LHS to RHS.
- Numpy files can be used as intermediate files between platforms. These binary files are memory efficient. Compression can be used to store sparse matrices.
- It is generally efficient to minimize the number of numpy functions being called in a given logic. Also, reducing the number of temporary variables improves the performance.

MCQ : Week 8

1. Can a numpy array be converted to a python list?
    1. **Yes**
    2. No
    3. Depends on the dimensions of the array.
    4. Depends on the data type of the array.
2. The size method is used to
    1. Find the dimensions along an axis
    2. **Find the number of elements**
    3. Both a and b
    4. None of the above
3. If **A** is a numpy array, how would the element at 3rd column and 6th row be accessed?
    1. A[6,3]
    2. A(6,3)
    3. A[2,5]
    4. **A[5,2]**
4. What is the output of the below code?

```
a = np.array([1,2,3,np.nan,5,6,7,np.nan])
a[~np.isnan(a)]
```

    1. Drops all missing values from a
    2. **Filters out nan values from a**
    3. **array([1., 2., 3., 5., 6., 7.])**
    4. All of the above.
5. Select the one that produces different output for a given array a:

```
a = np.array([2, 6, 1, 9, 10, 3, 27])
```

    1. a[np.where((a >= 5) & (a <= 10))]
    2. a[np.where(np.logical_and(a>=5, a<=10))]
    3. a[(a >= 5) & (a <= 10)]
    4. **All produce the same output.**
6. What is the output of the code below?

```
arr = np.arange(9).reshape(3,3)
arr[[1,0,2], :]
```

1. **Swaps the first and second row.**
2. Swaps the columns indexed 1 and 0.
3. Throws an error .